NUMERICAL SEMIGROUPS WITH A GIVEN SET OF PSEUDO-FROBENIUS NUMBERS

M. DELGADO, P. A. GARCÍA-SÁNCHEZ, AND A. M. ROBLES-PÉREZ

Abstract. The pseudo-Frobenius numbers of a numerical semigroup are those gaps of the numerical semigroup that are maximal for the partial order induced by the semigroup. We present a procedure to detect if a given set of integers is the set of pseudo-Frobenius numbers of a numerical semigroup and, if so, to compute the set of all numerical semigroups having this set as set of pseudo-Frobenius numbers.

1. Introduction

Let S be a numerical semigroup, that is, a cofinite submonoid of $(\mathbb{N}, +)$, where $\mathbb{N}$ stands for the set of nonnegative integers.

An integer x is said to be the *Frobenius number* of S (respectively, a *pseudo-Frobenius number* of S) if $x \notin S$ and $x + s \in S$, for all $s \in \mathbb{N} \setminus \{0\}$ (respectively, for all $s \in S \setminus \{0\}$).

Given a positive integer f, there exist numerical semigroups whose Frobenius number is f. One example of such a semigroup is the semigroup $\{0, f + 1, \rightarrow\}$ containing 0 and all the integers greater than f. There are several algorithms to compute all the numerical semigroups with a given Frobenius number (the fastest we know is based on [9]).

We denote by $\mathrm{F}(S)$ the Frobenius number of S and by $\mathrm{PF}(S)$ the set of pseudo-Frobenius numbers of S. The cardinality of $\mathrm{PF}(S)$ is said to be the *type* of S and is denoted by $\mathrm{t}(S)$.

A positive integer that does not belong to S is said to be a *gap* of S and an element of S that is not greater than $\mathrm{F}(S) + 1$ is said to be a *small element* of S. To denote the set $\mathbb{N} \setminus S$ of gaps of S we use $\mathrm{gaps}(S)$ and to denote the set of small elements of S we use $\mathrm{smalls}(S)$. Since a set of gaps must contain the divisors of all its members and a set of small elements must contain all multiples of its members (up to its maximum), it is clear that there are sets of positive integers that can not be the set of gaps or the set of small elements of a numerical semigroup. The set of gaps, as well as the set of small elements, completely determines the semigroup. Observe that when some elements or some gaps are known, others may be forced. For instance, a gap forces all its divisors to be gaps.

Let n be a positive integer and let $\mathrm{PF} = \{g_1, g_2, \ldots, g_{n-1}, g_n\}$ be a set of positive integers. Denote by $\mathcal{S}(\mathrm{PF})$ the set of numerical semigroups whose set of pseudo-Frobenius numbers is PF. When $n > 1$, $\mathcal{S}(\mathrm{PF})$ may clearly be empty. Moreover, when non-empty, it is finite. In fact, $\mathcal{S}(\mathrm{PF})$ consists of semigroups whose Frobenius numbers is the maximum of PF. Some questions arise naturally. Among them, we can consider the following.

Question 1. Find conditions on the set PF that ensure that $\mathcal{S}(\mathrm{PF}) \neq \emptyset$.

Question 2. Find an algorithm to compute $\mathcal{S}(\mathrm{PF})$.

Both questions have been solved for the case that the set PF consists of a single element (which must be the Frobenius number of a numerical semigroup; symmetric numerical semigroups) or when PF consists on an even positive integer f and $f/2$ (pseudo-symmetric numerical semigroups), see [1]. Moreover,

Question 1 was solved by Robles-Pérez and Rosales [7] in the case where PF consists of 2 elements (not necessarily of the form $\{f, f/2\}$).

The set $\mathcal{S}(\text{PF})$ can be computed by filtering those semigroups that have PF as set of pseudo-Frobenius numbers, from the numerical semigroups whose Frobenius number is the maximum of PF (cf. Example 12). Due, in part, to the possibly huge number of semigroups with a given Frobenius number, this is a rather slow procedure and we consider it far from being a satisfactory answer to Question 2.

Irreducible numerical semigroups with odd Frobenius number correspond with symmetric numerical semigroups, and those with even Frobenius number with pseudo-symmetric (see for instance [8, Chapter 3]). Bresinsky proved in [2] that symmetric numerical semigroups with embedding dimension four have minimal presentations of cardinality 5 or 3 (complete intersections). Symmetry of a numerical semigroup S translates to having $\{F(S)\}$ as set of pseudo-Frobenius numbers. Later Komeda, [6], was able to prove the same result for pseudo-symmetric numerical semigroups (though he used different terminology for this property; in this setting 3 does not occur since pseudo-symmetric are never complete intersections). A numerical semigroup S is pseudo-symmetric if its set of pseudo-Frobenius numbers is $\{F(S), F(S)/2\}$. It should be interesting to see the relationship with the type and the cardinality of a minimal presentation, and thus having tools to find semigroups with given sets of pseudo-Frobenius numbers becomes helpful. Watanabe and his students Nari and Numata are making some progress in the study of this relationship.

1.1. **Contents.** We present two different procedures to determine the set of all numerical semigroups with a given set of pseudo-Frobenius numbers. One exploits the idea of irreducible numerical semigroup. From each irreducible numerical semigroup we start removing minimal generators with certain properties to build a tree whose leafs are the semigroups we are looking for. The other approach is based on determining the elements and gaps of any numerical semigroup with the given set of pseudo-Frobenius numbers, obtaining in this way a list of "free" integers. We then construct a binary tree in which branches correspond to assuming that these integers are either gaps or elements.

We start this work with some generalities and basic or well known results and connections with the GAP [5] package numericalsgps [4] (Sections 2 and 3). Then we describe a procedure to compute forced integers (Sections 4 and 5). As computing forced integers is fast and leads frequently to the conclusion that there exists no semigroup fulfilling the condition of having the given set as set of pseudo-Frobenius numbers, this approach benefits from the work done in the preceding sections. The following section of the paper is devoted to the above-mentioned approach of constructing a tree with nodes lists of integers, which turns out to be faster most of the times than the one based on irreducible numerical semigroups. Nevertheless, besides being useful to compare results, the method using irreducible numerical semigroups is of theoretical importance so we decided to keep it, and it is given in Appendix A. The next section describes an algorithm that (increasing the number of attempts, if necessary) returns one numerical semigroup with the given set of pseudo-Frobenius numbers, when such a semigroup exists. Although it may be necessary to increase the number of attempts, usually does not require the computation of all the semigroups fulfilling the condition and is reasonably fast in practice (Section 8).

We give pseudo-code for the various algorithms that are used in the approach of lists of free integers. The pseudo-code given has an immediate translation to the GAP language, which is the programming language we used to implement these algorithms. We observe that it is a high level programming language. Note that we take advantage of the existence of the GAP package numericalsgps for computing with numerical semigroups. The names used for the functions described here are slightly different from the ones used in the package, since there longer names are required to be accurate with the variable names policy in GAP. Essentially, it will be a matter of adding a suffix to the names used for the package.

Many examples are given throughout the paper, some to illustrate the methods proposed, while others are included to motivate the options followed. In some of the examples we show the output obtained in a GAP session. These usually correspond to examples that are not suitable to a full computation just using a pencil and some paper; furthermore, indicative running times, as given by GAP, are shown (mainly in Section 7).

A new version of the numericalsgps package, including implementations of the algorithms developed in the present work, is released at the same time this work is made public. The implementations can

be checked (the software in question is open source; see the links in the references) and can be used for testing examples.

2. Generalities and basic results

Throughout the paper we will consider often a set PF = $\{g_1, g_2, \ldots, g_{n-1}, g_n\}$ of positive integers. We will usually require it to be ordered, that is, we will assume that $g_1 < g_2 < \cdots < g_{n-1} < g_n$. For convenience, we write PF = $\{g_1 < g_2 < \cdots < g_{n-1} < g_n\}$ in this case.

We denote by frob the maximum of PF and by type the cardinality of PF. Note that if $S \in \mathcal{S}(\text{PF})$, then frob = g_n = F(S) and type = n = t(S).

2.1. **Forced integers.** We say that an integer ℓ is a *gap forced by* PF or a PF-*forced gap* if ℓ is a gap of all the numerical semigroups having PF as set of pseudo-Frobenius numbers. In particular, if there is no semigroup with PF as set of pseudo-Frobenius numbers, then every nonnegative integer is a gap forced by PF. We use the notation $\mathcal{GF}(\text{PF})$ to denote the set of PF-forced gaps. In symbols: $\mathcal{GF}(\text{PF}) = \bigcap_{S \in \mathcal{S}(\text{PF})} \text{gaps}(S)$.

In a similar way, we say that an integer ℓ is an *element forced by* PF or a PF-*forced element* if ℓ is an element of all semigroups in $\mathcal{S}(\text{PF})$. We use the notation $\mathcal{EF}(\text{PF})$ to denote the set of (small) PF-forced elements. In symbols: $\mathcal{EF}(\text{PF}) = \bigcap_{S \in \mathcal{S}(\text{PF})} \text{smalls}(S)$. Note also that if $\mathcal{S}(\text{PF}) = \emptyset$, then $\mathcal{EF}(\text{PF}) = \mathbb{N}$.

The union of the PF-forced gaps and PF-forced elements is designated by PF-*forced integers*.

The following is a simple, but crucial observation.

Proposition 3. $\mathcal{S}(\text{PF}) \neq \emptyset$ if and only if $\mathcal{GF}(\text{PF}) \cap \mathcal{EF}(\text{PF}) = \emptyset$.

Proof. If $\mathcal{S}(\text{PF}) = \emptyset$, then all nonnegative integers are at the same time gaps and elements forced by PF. Conversely, assume that $\mathcal{S}(\text{PF}) \neq \emptyset$ and let $S \in \mathcal{S}(\text{PF})$. Then $\mathcal{GF}(\text{PF}) \cap \mathcal{EF}(\text{PF}) \subseteq \text{gaps}(S) \cap \text{smalls}(S) = \emptyset$. $\square$

Frequently the prefix PF is understood and we will abbreviate by saying just *forced gap*, *forced element* or *forced integer*.

Let G and E be, respectively, sets of forced gaps and forced elements. The elements $v \in \{1, \ldots, \text{frob}\}$ that do not belong to $G \cup E$ are said to be *free integers for* (G, E). When the pair (G, E) is understood, we simply call *free integer* to a free integer for (G, E).

2.2. **Well known results.** The partial order $\leq_S$ induced by the numerical semigroup S on the integers is defined as follows: $x \leq_S y$ if and only if $y - x \in S$. The following result is well known and will be used several times throughout this paper.

Lemma 4. [8, Lemma 2.19] *Let S be a numerical semigroup. Then*

(i) PF(S) = Maximals$_{\leq_S}(\mathbb{Z} \setminus S)$,
(ii) $x \in \mathbb{Z} \setminus S$ *if and only if* $f - x \in S$ *for some* $f \in \text{PF}(S)$.

It is well known that the type of a numerical semigroup S is upper bounded by the least positive element belonging to S, which is known as the *multiplicity* of S and it is denoted by m(S).

Lemma 5. [8, Corollary 2.23] *Let S be a numerical semigroup. Then* m(S) $\geq$ t(S) + 1.

2.3. **Initializing the set of forced gaps.** The maximality of the pseudo-Frobenius numbers of S with respect to $\leq_S$ means that they are incomparable with respect to this ordering. In particular, the difference of any two distinct pseudo-Frobenius numbers does not belong to S, that is,

(1) $$\{g_i - g_j \mid i, j \in \{1, \ldots, \text{t}(S)\}, i > j\} \subseteq \text{gaps}(S).$$

This is the underlying idea of the next result.

Lemma 6. *Let S be a numerical semigroup and suppose that* PF(S) = $\{g_1 < g_2 < \cdots < g_{n-1} < g_n\}$, *with $n > 1$. Let $i \in \{2, \ldots, \text{t}(S)\}$ and $g \in \langle \text{PF}(S) \rangle$ with $g < g_i$. Then $g_i - g \in \text{gaps}(S)$.*

Proof. Assume that $g = g_{i_1} + \cdots + g_{i_k}$ for some $k \in \mathbb{N}$. We proceed by induction on k. The case $k = 1$ is given by (1). Assume that the result holds for $k - 1$ and let us prove it for k. If $g_i - g \in S$, then $g_{i_k} + (g_i - g) \in S$ by definition of pseudo-Frobenius number. It follows that $g_i - (g_{i_1} + \cdots + g_{i_{k-1}}) \in S$, contradicting the induction hypothesis. $\qquad\square$

Remark 7. Lemma 5 implies that $\{x \in \mathbb{N} \mid 1 \le x \le \mathrm{t}(S)\} \subseteq \mathrm{gaps}(S)$. Hence $\{1, \ldots, n\} \subseteq \mathcal{GF}(\mathrm{PF})$.

As the pseudo-Frobenius numbers of S are gaps of S and any positive divisor of a gap must be a gap also, we conclude that the set of divisors of

$$\mathrm{PF}(S) \cup \{x \in \mathbb{N} \mid 1 \le x \le \mathrm{t}(S)\} \cup \{g_i - g \mid i \in \{2, \ldots, \mathrm{t}(S)\}, g \in \langle \mathrm{PF}(S) \rangle, g_i > g\}$$

consists entirely of gaps of S.

Consider the set

$$\mathrm{PF} \cup \{x \in \mathbb{N} \mid 1 \le x \le n\} \cup \{g_i - g \mid i \in \{2, \ldots, n\}, g \in \langle \mathrm{PF} \rangle, g_i > g\}$$

and denote by $\mathrm{sfg}(\mathrm{PF})$ the set of its divisors (as we are only considering positive divisors, in what follows we will not include this adjective). If S is a numerical semigroup such that $\mathrm{PF}(S) = \mathrm{PF}$, we deduce that $\mathrm{sfg}(\mathrm{PF}) \subseteq \mathrm{gaps}(S)$. We have proved the following result for the case where there is a numerical semigroup S such that $\mathrm{PF}(S) = \mathrm{PF}$. If no such semigroup exists, then $\mathcal{GF}(\mathrm{PF}) = \mathbb{N}$ and the result trivially holds.

Corollary 8. *Let* PF *be a set of positive integers. Then* $\mathrm{sfg}(\mathrm{PF}) \subseteq \mathcal{GF}(\mathrm{PF})$.

We use the terminology *starting forced gap for* PF to designate any element of $\mathrm{sfg}(\mathrm{PF})$, since $\mathrm{sfg}(\mathrm{PF})$ is the set we start with when we are looking for forced gaps. In Subsection 4.1 we provide pseudo-code for a function to compute starting forced gaps.

2.4. **Initial necessary conditions.** Let $n > 1$ be an integer and let $\mathrm{PF} = \{g_1 < \cdots < g_n\}$ be a set of positive integers.

Lemma 9. *Let* S *be a numerical semigroup such that* $\mathrm{PF}(S) = \mathrm{PF}$. *Let* $i \in \{2, \ldots, n\}$ *and* $g \in \langle \mathrm{PF} \rangle \setminus \{0\}$ *with* $g < g_i$. *Then there exists* $k \in \{1, \ldots, n\}$ *such that* $g_k - (g_i - g) \in S$.

Proof. Lemma 6 assures that $g_i - g \notin S$. The conclusion follows from Lemma 4. $\qquad\square$

By choosing $i = n$ in the above result, there exists $k \ne n$ such that $g_k - (g_n - g_1) \ge 0$ and $g_k - (g_n - g_1) \notin \{1, \ldots, \mathrm{type}\}$. (Note that $k = n$ would imply $g_1 \in S$, which is impossible.) But then $g_{n-1} - (g_n - g_1) \ge 0$, since $g_{n-1} \ge g_k$ for all $k \in \{1, \ldots, n-1\}$. We have thus proved the following corollary.

Corollary 10. *Let* S *be a numerical semigroup such that* $\mathrm{PF}(S) = \mathrm{PF}$. *Then* $g_1 \ge g_n - g_{n-1}$.

The computational cost of testing the condition $g_1 \ge g_n - g_{n-1}$ obtained in Corollary 10 is negligible and should be made before calling any procedure to compute $\mathcal{S}(\mathrm{PF})$, avoiding in many cases an extra effort that would lead to the empty set.

Other conditions of low computational cost would also be useful. Since $g_{n-1} - (g_n - g_1) \ge 0$, one could be tempted to ask whether replacing g_1 by g_2 one must have $g_{n-1} - (g_n - g_2) \notin \{1, \ldots, \mathrm{type}\}$ (since $\{1, \ldots, \mathrm{type}\}$ consists of gaps). The following example serves to rule out this one that seems to be a natural attempt.

Example 11. Let $S = \langle 8, 9, 10, 11, 13, 14, 15 \rangle$. One can check easily that $\mathrm{PF}(S) = \{5, 6, 7, 12\}$. For $\mathrm{PF} = \{5, 6, 7, 12\}$, we have $g_{n-1} - (g_n - g_2) = 7 - (12 - 6) = 1 \in \{1, \ldots, 4\} = \{1, \ldots, \mathrm{type}\}$.

Later, in Subsection 4.3, we give an extra condition, which is based on forced integers.

3. CONNECTIONS WITH GAP AND THE PACKAGE NUMERICALSGPS

When developing the present algorithms we had one implementation in mind. As already referred, it was to be made in the GAP programming language, benefiting also of the package numericalsgps. In fact the implementation accompanied the development of the algorithms, with mutual benefits.

To understand the paper no previous familiarity with GAP is assumed. We give some examples to get the familiarity needed to fully understand the pseudo-code. Many of the terms used for the pseudo-code presented are borrowed from the GAP programming language, which is a high level one. The terms *Union, Length, Difference* are used with the meaning they have in GAP, which is clear.

Like what is done in GAP, for an unary function that applied to an argument *arg* returns an expression *expr*, we write

$$arg \rightarrow expr.$$

This shorthand for writing a function is used in our pseudo-code. Next we briefly explain the meaning of some of the terms used. For a not so brief explanation, complete definitions and plenty of examples, GAP's manual should be consulted.

Here *list* is a list (of integers or of numerical semigroups) and *func* is an unary function (that applies to the objects in *list*).

- List(*list*, *func*) returns a new list *new* of the same length as *list* such that $new[i] = func(list[i])$.
- Filtered(*list*, *func*) returns a new list that contains those elements of *list* for which *func* returns *true*. The order of the elements in the result is the same as the order of the corresponding elements of this list.
- One can use Set(*list*) (which is a synonym of SSortedList ("strictly sorted list")) to get a list that is duplicate free and sorted (for some order).
- AddSet(*list*, *el*) adds the element *el* to the set *list*.
- First(*list*, *func*) returns the first element of *list* for which the unary function *func* returns *true*. If func returns false for all elements of list, then First returns *fail*.
- ForAll(*list*, *func*) tests whether *func* returns *true* for all elements in *list*.
- ForAny(*list*, *func*) tests whether *func* returns *true* for at least one element in *list*.
- IsRange(*list*) detects if the argument is an interval of integers.

We also use the following abbreviations. Let X be a list of integers and P a list of positive integers.

- PosInt(X), an abbreviation of Filtered(X,IsPosInt), returns the positive integers of X.
- Divisors(P), an abbreviation of Union(List(P,DivisorsInt)), returns the divisors of the elements of P: it first computes the lists of divisors for each element in P, and then takes the union of all of them.

The numericalsgps package also influences our pseudo-code. In some cases we use directly the names of the available functions, but in some cases we use shorthands, which are intended to turn the pseudo-code more readable.

One of the functions we use is RepresentsGapsOfNumericalSemigroup, which, for a given set X of positive integers returns *true* in case there exists a numerical semigroup S such that gaps$(S) = X$, and returns *false* otherwise.

The functions to produce numerical semigroups used here are NumericalSemigroup, which is used when generators are given, and NumericalSemigroupByGaps, which, for an input X, returns a numerical semigroup whose set of gaps is X, when such a semigroup exists.

As a shorthand for SmallElements(S), which gives the elements of the numerical semigroup S that are nor greater than F$(S) + 1$, we use simply smalls, which agrees with the notation already introduced. Closure(elts,frob) is a shorthand for

```
NumericalSemigroup(Union(elts,[frob+1..frob+Minimum(elts)]))
```

which gives the least numerical semigroup containing the set elts of positive integers and having the largest possible Frobenius number not greater than frob. Note that the minimum of elts is greater than or equal to the multiplicity of the semigroup. Therefore, the union considered ensures that the semigroup contains all integers that are greater than the number frob given (aiming to be an upper bound for the Frobenius number of the semigroup constructed, although in cases where the elements elts define a numerical semigroup, it may have smaller Frobenius number).

As in GAP, comments start with the character #.

As an example on how we will present function/algorithms in this manuscript, we write the function Closure.

```
Closure(elts,frob)                        # the least numerical semigroup containing elts and
                                          # whose Frobenius number is not greater than frob
  return NumericalSemigroup (Union (elts,[frob +1..frob +Minimum (PositiveInt (elts))]));
```
Function Closure

The following example is just a GAP session that is intended to illustrate how to compute the set $\mathcal{S}(\text{PF})$ by filtering those semigroups that have PF as set of pseudo-Frobenius numbers, from the numerical semigroups whose Frobenius number is the maximum of PF. This process was mentioned in the introduction.

Throughout the examples, we use a simple way provided by GAP to give a rough idea of the time spent: time is a global variable that stores the time in milliseconds the last command took.

Example 12. We illustrate how to compute the set of numerical semigroups having $\{19, 29\}$ as set of pseudo-Frobenius numbers. Double semicolon in GAP inhibits the output.

```
gap> pf := [19,29];;
gap> nsf29 := NumericalSemigroupsWithFrobeniusNumber(29);;time;
31372
gap> Length(nsf29);
34903
gap> nspf1929 := Filtered(nsf29, s -> PseudoFrobeniusOfNumericalSemigroup(s) = pf);;
gap> time;
2540
gap> Set(nspf1929,MinimalGeneratingSystem);
[ [ 3, 22, 32 ], [ 6, 9, 16, 26 ], [ 7, 9, 17 ], [ 8, 9, 14 ], [ 8, 9, 15, 22, 28 ],
[ 9, 12, 13, 14 ], [ 9, 12, 13, 15, 23 ], [ 9, 12, 14, 16, 22 ],
[ 9, 12, 15, 16, 22, 23, 26 ], [ 9, 13, 14, 17, 21, 24, 25 ],
[ 9, 13, 15, 17, 21, 23, 25 ], [ 9, 14, 16, 17, 21, 22, 24 ],
[ 9, 15, 16, 17, 21, 22, 23, 28 ] ]
```

Prior to the obtention of the procedures that are the object of study of the present paper, we got the necessary insight through detailed analysis of many examples with PF consisting of small numbers (less than 29, say). Example 12 illustrates that this can be easily done by using the numericalsgps package. When PF consists of small numbers the time spent is acceptable.

The implementation of our algorithm (available in Version 0.99 of the numericalsgps package) is much faster. It is even faster than just filtering among the numerical semigroups with a given set of pseudo-Frobenius numbers.

Example 13. To illustrate, we continue the GAP session started in Example 12.

```
gap> new := NumericalSemigroupsWithPseudoFrobeniusNumbers(pf);;time;
29
gap> Set(new,MinimalGeneratingSystem)=Set(nspf1929,MinimalGeneratingSystem);
true
```

Visualization of images obtained by using the intpic [3] GAP package have also helped us to improve our understanding of the problem and to get the necessary intuition.

The production of Figure 1 takes less than two minutes in our laptop (using numericalsgps and intpic packages). Taking into account that there are 34903 numerical semigroups with Frobenius number 29, one can consider the performance of the implementations satisfactory. In order to explain the meaning of the colors used in this picture to highlight some integers, we anticipate the results shown in Example 23. For PF = $\{19, 29\}$, we have that $\{1, 2, 4, 5, 10, 11, 19, 20\}$ consists of forced gaps and that $\{0, 9, 18, 24, 25, 27, 28, 30\}$ consists of forced elements. To the elements in each of these sets, as well as the ones in the sets of minimal generators, is assigned one color (*red* corresponds to pseudo-Frobenius numbers, *blue* to minimal generators, *green* to elements, *cyan* to forced gaps, and *magenta* to forced elements;
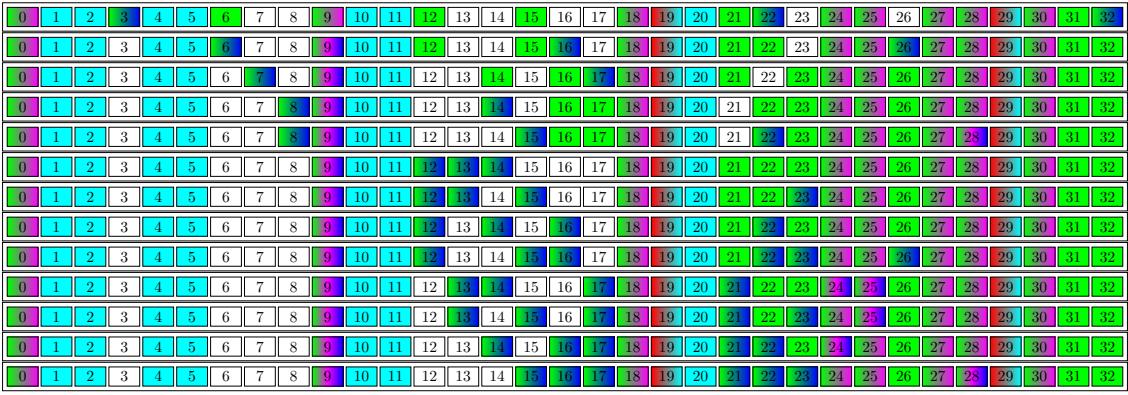
FIGURE 1. The numerical semigroups with pseudo-Frobenius numbers $\{19, 29\}$.

in a black and white visualization of this manuscript this will correspond with different gray tonalities). For integers that belong to more that one set, gradient colors are assigned.

4. Integers forced by pseudo-Frobenius numbers - some preliminary procedures

In this section we give pseudo-code for several functions implementing results given in Section 2 and some others needed later in Section 5.

Once more, PF is a fixed set $\{g_1 < g_2 < \cdots < g_{n-1} < g_n\}$ of positive integers; frob stands for g_n and type stands for n.

4.1. **Forced gaps.** The function StartingForcedGaps returns the integers considered in Subsection 2.3, which we called starting forced gaps. Recall (Corollary 8) that these have to be gaps of all the numerical semigroups having PF as set of pseudo-Frobenius numbers. The justifications for Lines 1 and 2 are given by Remark 7 and Lemma 6, respectively. Line 3 is justified by Lemma 4(ii): when it is detected a gap that had to be an element (forced by exclusion), there exists a contradiction and the function returns fail. This proves the following proposition.

Proposition 14. *Let OUT be the output of the function StartingForcedGaps for the input* PF. *If OUT is not* fail, *then it consists of gaps of any numerical semigroup* $S \in \mathcal{S}(PF)$.

```
StartingForcedGaps(PF)
1  fg := Union([1..type], PF);                              # uses the fact m(S) ≥ t(S)+1 (Remark 7)
   closures := List([1..type − 1], i → smalls(Closure({PF[1], PF[2], ..., PF[i]}, PF[i + 1])));
   diffs := ∅;
2  for i ∈ [2..type] do
       diffs := Union(diffs, PositiveInt(PF[i] − closures[i − 1]));           # see Lemma 6
   fg := Divisors(Union(fg, diffs));
   # Detect possible contradiction given by Lemma 4(ii)
3  for x ∈ Difference(fg, PF)) do
       if PositiveInt(Difference(PF − x, fg)) = ∅ then
           return fail
   return fg;
```
Function StartingForcedGaps

Example 15. Let PF = $\{16, 29\}$. One can easily check that the set of starting forced gaps is $\{1, 2, 4, 8, 13, 16, 29\}$. The corresponding GAP session:

```
gap> pf := [16,29];;
gap> g := StartingForcedGapsForPseudoFrobenius(pf);
[ 1, 2, 4, 8, 13, 16, 29 ]
```

The function FurtherForcedGaps is used to determine forced gaps when some gaps and some elements of a numerical semigroup are known. The justification for the fact that the output of FurtherForcedGaps consists of gaps (unless there is an element that also had to be a gap, in which case it returns *fail*) is the following: if $f - e$ and e are elements of a semigroup then $e + f - e = f$ belongs to the semigroup. In particular, if e is an element and f is a gap, then $f - e$ is either negative or a gap. This proves the following proposition.

Proposition 16. *Let OUT be the output of the function FurtherForcedGaps for the input* (fg,fe), *with* fg *and* fe *consisting of* PF-*forced gaps and* PF-*forced elements, respectively. If OUT is* fail, *then* $\mathcal{S}(\mathrm{PF}) = \emptyset$. *Otherwise, OUT consists of gaps of any numerical semigroup* $S \in \mathcal{S}(\mathrm{PF})$.

```
FurtherForcedGaps(fg,fe)                          # fg and fe consist of gaps and elements,
                                                  # respectively
  ng := Divisors(PositiveInt(Union(List(fg, g → g − fe))));
1 if Intersection(fe, ng) = ∅ then
    return Union(fg, ng);
  else
    return fail;
```

Function FurtherForcedGaps

4.2. **Forced elements.** We use two ways to get new forced elements. One of these ways makes use of Lemma 4(ii). We refer to the elements obtained in this way as *elements forced by exclusion*. Another way makes use of the following lemma, which tells us that small gaps force elements that are close to the maximum of PF. Sometimes we refer to them by using the more suggestive terminology *big forced elements*.

Lemma 17. *Let m be the multiplicity of a numerical semigroup S and let i be an integer such that $1 \le i < m$. Then either* $\mathrm{F}(S) - i \in S$ *or* $\mathrm{F}(S) - i \in \mathrm{PF}(S)$.

Proof. It suffices to observe that, as $i < m$, one has that $\mathrm{F}(S) - i + m > \mathrm{F}(S)$, and consequently $\mathrm{F}(S) - i + m \in S$. The result follows immediately from the definition of pseudo-Frobenius numbers. □

We observe that in the function FurtherForcedElements it is used that PF is precisely the set of pseudo-Frobenius numbers; otherwise there is no guarantee that the output consists of forced elements.

Let us now prove the correctness of this function. Justification for the the result produced by the cycle starting in Line 2 is given by the direct implication of Lemma 4(ii). For the cycle starting in Line 3 is given by the reverse implication of the same lemma. (The integers known to be forced gaps are assumed to be gaps.)

Justification for Line 1 follows from Lemma 17, since the m appearing there is smaller than or equal to the multiplicity.

We have then the following proposition.

Proposition 18. *Let OUT be the output of the function FurtherForcedElements for the input* (fg,fe), *with* fg *and* fe *consisting of* PF-*forced gaps and* PF-*forced elements, respectively. If OUT is "fail", then* $\mathcal{S}(\mathrm{PF}) = \emptyset$. *Otherwise, OUT consists of elements of any numerical semigroup* $S \in \mathcal{S}(\mathrm{PF})$.

```
  FurtherForcedElements(fg,fe)    # fg and fe consist of gaps and elements, respectively
  # Big forced elements
  m := First(Integers, n → n > 0 and n ∉ fg);          # least integer that is not a forced gap
1 be := Difference(frob − {1,...,m−1}, PF);
  # Elements forced by exclusion
  ee := ∅;
2 for x ∈ fg do
  │  filt := Filtered(PF, f → not((f − x ∈ fg) or (f − x < 0)));
  │  if Length(filt) = 1 and filt[1] − x ∉ ee then
  │  └  AddSet(ee, filt[1] − x);

  candidates := Difference([1..frob − 1], Union(fg, Union(fe, ee)));
3 for x ∈ candidates do
  │  if PositiveInt(Difference(PF − x, fg)) = ∅ then
  │  └  AddSet(ee, x);

  ne := Union(fe, ee, be);
4 if Intersection(fg, ne) = ∅ then
  │  return ne;
  else
  │  return fail;
```

Function FurtherForcedElements

4.3. **A condition based on forced integers.** When searching for forced integers, one should pay attention to the existence of possible contradictions.

Example 19. Let $PF = \{4, 9\}$. Taking divisors and the difference $9 − 4$, one immediately sees that the set of starting forced gaps contains $\{1, 2, 3, 4, 5, 9\}$. But then 5 appears as a forced gap and as a (big) forced element. This is a contradiction which shows that $\{4, 9\}$ cannot be the set of pseudo-Frobenius numbers of a numerical semigroup.

According to Proposition 3, the only that we need to take into account is that the set of forced gaps is disjoint from the set of forced elements.

At the end of the functions (Line 1 in function FurtherForcedGaps, and Line 4 in function FurtherForcedElements), possible contradictions are detected.

5. A procedure to compute integers forced by pseudo-Frobenius numbers

Let $PF = \{g_1, g_2, \ldots, g_{n-1}, g_n\}$ be a set of positive integers. The aim of this section is to give a procedure to compute some elements of $\mathcal{GF}(PF)$ and $\mathcal{EF}(PF)$.

We start with a subsection that contains a procedure that makes use of the functions given in Section 4, which implement some well-known facts recalled in Section 2.

We then devote a subsection to what we call admissible integers. This will yield another procedure (in Subsection 5.3), which, at the cost of increasing the execution time, may find more forced integers.

The main algorithm of the present paper (Algorithm 3) would work as well by using only the quick version. Our experiments led us to consider the option of using the slower version once, and then use the quick version inside a recursive function that is called by the main algorithm.

5.1. **A quick procedure to compute forced integers.** The correctness of Algorithm 1 (SimpleForced-Integers) follows from Propositions 16 and 18 which state the correction of the functions considered in Section 4.

Theorem 20. *Algorithm 1 correctly produces the claimed output.*

Input : g, e, where g, e are sets of PF-forced gaps and PF-forced elements, respectively.
Output: [fg, fe], where fg ⊇ g is a set of PF-forced gaps and fe ⊇ e is a set of PF-forced elements, or
         fail, when some inconsistency is discovered

fg := ShallowCopy(g);   # used to store new forced elements and gaps
fe := ShallowCopy(e);   # without creating conflicts in the memory
repeat
    changes := false;
1   *gaps* := FurtherForcedGaps(fg,fe);
    if *gaps* = fail **then**
    │ **return** fail
    else
       if *gaps* ≠ fg **then**
          changes := true;
          fg := *gaps*;
2   *elts* := FurtherForcedElements(fg,fe);
    if *elts* = fail **then**
    │ **return** fail
    else
       if *elts* ≠ fe **then**
          changes := true;
          fe := *elts*;
until not *changes*;
return [fg,fe];

Algorithm 1: SimpleForcedIntegers

Note that the forced elements returned by Algorithm 1 (the list in the second component) are obtained by applying the function Closure to some set. Therefore we observe the following.

Remark 21. The second component of the list returned by Algorithm 1 is the set of small elements of a numerical semigroup.

Example 22. Let us see how the SimpleForcedIntegers works for PF = {16, 29}. We already know (Example 15) that the set of starting forced gaps is $sfg = \{1, 2, 4, 8, 13, 16, 29\}$.
   Let us now make a call to SimpleForcedIntegers with input sfg, [ ].
   The first passage in Line 1 does not produce any new integer.
   The first passage in Line 2 produces

$$\{0, 3, 6, 9, 12, 15, 18, 21, 24, 25, 27, 28, 30\}$$

as current forced elements. Observe that 3 is forced by exclusion (note that $3 = 16 - 13$; also, $29 - 13 = 16$ and 16 is a forced gap); 25 is also forced by exclusion (note that $16 - 25 < 0$ and $29 - 25 = 4$ is a forced gap). Also, 21 is forced by exclusion, but for now we do not need to worry with the multiples of 3, because these will appear when taking the closure.
   The second passage in Line 1 produces

$$\{1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 20, 23, 26, 29\}$$

as current forced gaps. To check it, observe that $29 - 3 = 26$, $29 - 6 = 23$, $29 - 9 = 20$, etc. are forced gaps. But then, 10 and 5 are (forced to be) gaps.
   No further forced elements appear. In fact, the union of the sets of forced gaps and of forced elements is {0,..., 30}.
   Therefore, all positive integers less than 29 are forced. One can check that the closure of the set of forced elements does not produce new forced elements, thus it is the set of small elements of a numerical

semigroup. Also, one can check that no forced gap outside {16, 29} is a pseudo-Frobenius number, thus one may conclude that there exists exactly one numerical semigroup S such that PF(S) = {16, 29}.

This example illustrates that more than one passage through the *repeat-until* loop of the algorithm SimpleForcedIntegers may be needed.

Example 23. In this example we present a GAP session, now for PF{19, 29}.

```
gap> pf := [19,29];;
gap> sfg := StartingForcedGapsForPseudoFrobenius(pf);
[ 1, 2, 5, 10, 19, 29 ]
gap> SimpleForcedIntegersForPseudoFrobenius(sfg,[],pf);
[ [ 1, 2, 4, 5, 10, 11, 19, 20, 29 ], [ 0, 9, 18, 24, 25, 27, 28, 30 ] ]
```

Recall that the names used in our package are longer than the ones used in this manuscript, so for instance, SimpleForcedIntegersForPseudoFrobenius is the name we have used in our package for Algorithm 1. This example is related to Figure 1, where forced integers are highlighted.

Example 24. Let us now apply the algorithm to PF = {15, 20, 27, 35}. Again, we will use GAP to help us in doing the calculations (which can be easily confirmed by hand).

```
gap> pf := [ 15, 20, 27, 35 ];;
gap> sfg := StartingForcedGapsForPseudoFrobenius(pf);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20, 27, 35 ]
```

We immediately get that {25, 26, 28, 29, 30, 31, 32, 33, 34} consists of forced big elements. And one can observe that 19 and 23 are forced by exclusion. This leads to the obtention of $35 - 19 = 16$ as forced gap. No other forced elements are obtained, which agrees with the following continuation of the GAP session:

```
gap> SimpleForcedIntegersForPseudoFrobenius(sfg,[],pf);
[ [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 16, 20, 27, 35 ],
  [ 0, 19, 23, 25, 26, 28, 29, 30, 31, 32, 33, 34, 36 ] ]
```

5.2. **Admissible integers.** Let G and E respectively be sets of PF-forced gaps and PF-forced elements, and let v be a free integer for (G, E). We say that v is *admissible for* (G, E) if Algorithm 1 when applied to $(G, E \cup \{v\})$ does not return *fail*. Otherwise, we say that v is *non-admissible for* (G, E). Thus, v is non-admissible implies that v can not be an element of any semigroup in $\mathcal{S}$(PF) and therefore is a gap of all semigroups in $\mathcal{S}$(PF), that is, is a forced gap.

Lemma 25. *Let G and E be sets of forced gaps and forced elements, respectively. Let v be free for (G, E). If v is non-admissible for (G, E), then v is a PF-forced gap.*

Observe that a semigroup generated by admissible elements for some pair (G, E) consists of admissible elements for (G, E).

The function NonAdmissible, with input a pair of sets of forced gaps and forced elements, returns non-admissible integers, which, by Lemma 25, are new forced gaps. This function is called by Algorithm 2, a not so quick procedure to compute forced integers.

Proposition 26. *Let OUT be the output of the function NonAdmissible for the input (fg,fe), with fg and fe consisting of PF-forced gaps and PF-forced elements, respectively. Then OUT is a set of integers that are non-admissible for (fg,fe).*

Example 27. Let $PF = \{11, 22, 23, 25\}$.

```
gap> pf := [ 11, 22, 23, 25 ];;
gap> sfg := StartingForcedGapsForPseudoFrobenius(pf);
[ 1, 2, 3, 4, 5, 6, 7, 11, 12, 14, 22, 23, 25 ]
```

By using the function SimpleForcedIntegers one obtains the following.

```
gap> SimpleForcedIntegersForPseudoFrobenius(sfg,[],pf);
[ [ 1, 2, 3, 4, 5, 6, 7, 11, 12, 14, 22, 23, 25 ],
  [ 0, 18, 19, 20, 21, 24, 26 ] ]
```

```
NonAdmissible(fg,fe)                # fg and fe consist of forced gaps and forced elements,
respectively
admissible := ∅;
totest := Difference([1..frob],Union(fg,fe));                    # start with the free integers
while totest ≠ ∅ do
    v := totest[1];
    pnfce := SimpleForcedIntegers(fg,Union(fe,[v]),PF);
    if pnfce ≠ fail then
        admissible := Union(admissible,pnfce[2]);
        totest := Difference(totest,admissible);
    else
        totest := Difference(totest,[v]);
return Difference([1..frob],admissible);
```

Function NonAdmissible

That $\{1,2,3,4,5,6,7,11,12,14,22,23,25\}$ consists of forced gaps and that the set $\{0,18,19,20,21,24,26\}$ can easily be confirmed by hand.

Let us now check that 15 is non-admissible. If it was an element of a semigroup $S \in \mathcal{S}(PF)$, then $10(=25-15)$ and $8(=23-15)$ would be gaps of S. But then 17 is a big element, $13(=23-10)$ is forced by exclusion (note that $25-10=15$, $22-10=12$ and $11-10=1$ are gaps) and $9(=23-14)$ is forced by exclusion too ($25-14=11$, $22-14=8$ are gaps and $11-14<0$). This is not possible, since $13+9=22$ is a gap. Therefore 15 is non-admissible.

5.3. **A not so quick procedure to compute forced integers.** Algorithm 2 is our procedure to compute forced integers that produces the best result in terms of the number of forced integers encountered. Besides Algorithm 1, it makes use of the function NonAdmissible.

Remark 28. It is a consequence of Remark 21 that the second component of the list returned by Algorithm 2 is the set of small elements of a numerical semigroup.

```
Input  : PF
Output: fail if some inconsistency is discovered; otherwise, returns [fg, fe ], where fg and fe are sets
         of forced gaps and forced elements, respectively.
if type = 1 then
    return [Divisors(frob),[0,frob +1]];
fints := SimpleForcedIntegers(StartingForcedGaps(PF),[ ]);
if fints = fail then
    return fail
else
    if IsRange (Union(fints)) then
        return fints;
nad := NonAdmissible(fints[1],fints[2]);
newgaps := Difference(nad,fints[1]);
return SimpleForcedIntegers(Union(newgaps,fints[1]),fints[2]);
```

Algorithm 2: ForcedIntegers

The correctness of Algorithm 2 follows from Proposition 26 and Theorem 20.

Theorem 29. *Algorithm 2 correctly produces the claimed output.*

pseudo Frobenius numbers	Time		# F. Gaps		# F. Elements	
	QV	NV	QV	NV	QV	NV
[11, 22, 23, 25]	2	7	13	14	7	7
[17, 27, 28, 29]	2	5	16	17	8	10
[17, 19, 21, 25, 27]	2	9	15	16	8	8
[15, 20, 27, 35]	2	10	16	16	13	13
[12, 24, 25, 26, 28, 29]	3	11	18	22	6	9
[145, 154, 205, 322, 376, 380]	47	1336	82	85	52	54
[245, 281, 282, 292, 334, 373, 393, 424, 432, 454, 467]	129	2075	116	116	53	53
[223, 434, 476, 513, 549, 728, 828, 838, 849, 953]	213	5866	300	318	221	253
[219, 437, 600, 638, 683, 779, 801, 819, 880]	205	4838	219	224	153	161
[103, 110, 112, 137, 160, 178, 185]	25	262	50	51	29	31

Table 1. Execution times for computing forced integers.

5.4. **Examples and execution times.** Algorithm 1 can be used as a quick way to compute forced integers. In fact, when called with the starting forced gaps and the empty set of elements, can be seen as a quick version of Algorithm 2. We use

$$\mathtt{ForcedIntegers\_QV(PF)}$$

as a shorthand for

$$\mathtt{SimpleForcedIntegers(StartingForcedGaps\ (PF),[\ ])}$$

In the following example we use the names for the functions in the current implementation.

Example 30. This example is meant to illustrate the difference between applying the quick and the normal version of the algorithm.

```
gap> pf := [ 103, 110, 112, 137, 160, 178, 185 ];;
gap> fQ := ForcedIntegersForPseudoFrobenius_QV(pf);;time;
25
gap> fN := ForcedIntegersForPseudoFrobenius(pf);;time;
262
gap> Length(fQ[1]);Length(fN[1]);Length(fQ[2]);Length(fN[2]);
50
51
29
31
```

We have used the internal names in the package ForcedIntegersForPseudoFrobenius_QV and ForcedIntegersForPseudoFrobenius for ForcedIntegers_QV and ForcedIntegers, respectively.

Table 1 collects some information concerning some execution times (as given by GAP) and the number of forced gaps and of forced elements both using Algorithm 1 (identified as QV (which stands for *quick version*)) and Algorithm 2 (identified as NV (which stands for *normal version*)). We observe that the execution times when using the quick version remain relatively small, even when the Frobenius number is large.

Failure is usually detected very quickly, as should be clear and Table 2 somehow confirms.

As one could expect, there are examples where failure is not detected with the quick version.

```
gap> pf := [ 25, 29, 33, 35, 38, 41, 46 ];
[ 25, 29, 33, 35, 38, 41, 46 ]
gap> ForcedIntegersForPseudoFrobenius(pf);
fail
gap> ForcedIntegersForPseudoFrobenius_QV(pf);
[ [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16, 17, 19, 21, 23, 25, 29, 33, 35,
```

pseudo Frobenius numbers	Time	
	QV	NV
[18, 42, 58, 88, 94]	6	6
[20, 27, 34, 35, 37, 42, 48, 80]	2	3
[30, 104, 118, 147, 197, 292, 298, 315, 333, 384, 408]	32	43
[36, 37, 219, 233, 304, 410, 413, 431, 438, 458]	35	32
[89, 411, 446, 502, 557, 600, 605, 631, 636, 796, 801, 915]	223	233
[56, 134, 136, 137, 158, 248, 277, 373, 383, 389, 487, 558, 566, 621, 691, 825, 836]	103	113

TABLE 2. When failure occurs...

```
 38, 41, 46 ], [ 0, 30, 34, 36, 37, 39, 40, 42, 43, 44, 45, 47 ] ]
gap> pf := [ 22, 23, 24, 25, 26 ];
[ 22, 23, 24, 25, 26 ]
gap> ForcedIntegersForPseudoFrobenius_QV(pf);
[ [ 1, 2, 3, 4, 5, 6, 8, 11, 12, 13, 22, 23, 24, 25, 26 ], [ 0, 20, 21, 27 ] ]
gap> ForcedIntegersForPseudoFrobenius(pf);
fail
```

On the other hand, we have not been able to detect any set PF candidate to be the set of pseudo-Frobenius numbers of a numerical semigroup that passes the normal version and such that $\mathcal{S}(\mathrm{PF}) = \emptyset$. Despite the various millions of tests made, we do not feel comfortable on leaving it as a conjecture; we leave it as a question instead.

Question 31. If Algorithm 2 with input PF does not return *fail*, then $\mathcal{S}(\mathrm{PF}) \neq \emptyset$?

We observe that in view of the execution times illustrated in Table 1, a positive answer to Question 31 would imply that Algorithm 2 constitutes a satisfactory answer to Question 1.

6. AN APPROACH BASED ON FORCED INTEGERS

In this section we present our main algorithm (Algorithm 3), which computes $\mathcal{S}(\mathrm{PF})$. Its correctness is stated in Theorem 39, whose proof is built from almost all the results preceding it in the paper.

After considering some initial cases, the algorithm makes a call to RecursiveDepthFirstExploreTree which is a recursive function used to construct a tree whose nodes are labeled by pairs (X, Y) where X is a list of forced gaps and Y is a list of forced elements. Thus we implicitly have lists of free integers in each node: the complement of $X \cup Y$ in the set $U = \{1, \ldots, g_n\}$, where $\mathrm{PF} = \{g_1 < \cdots < g_n\}$. Nodes with an empty set of free integers are the leafs in our tree.

A node (X, Y) such that there exists a numerical semigroup $S \in \mathcal{S}(\mathrm{PF})$ for which $X \subseteq \mathrm{gaps}(S)$ and $Y \subseteq \mathrm{smalls}(S)$ is said to be PF-*admissible*, or simply *admissible*, if PF is understood. A node that is not admissible is called a *dead node*.

Remark 32. The knowledge of some forced integers allows us to identify immediately some dead nodes: if (G, E) is such that G consists of forced gaps and E consists of forced elements, then any node (X, Y) such that $X \cap E \neq \emptyset$ or $Y \cap G \neq \emptyset$ is a dead node.

Remark 33. Let (X, Y) be a leaf that is not a dead node. It follows from the construction (see Remarks 21 and 28) that there exists a numerical semigroup S such that $(\mathrm{gaps}(S), \mathrm{smalls}(S)) = (X, Y)$.

Remark 34. Observe that if $\mathrm{PF} = \{g_1\}$ or $\mathrm{PF} = \{g_1/2 < g_1\}$, then the set of numerical semigroups with pseudo-Frobenius numbers PF corresponds with the set of irreducible numerical semigroups having Frobenius number g_1; see Appendix A. In this case we will use the fast procedure presented in [1].

6.1. **The recursive construction of a tree.** A naive idea is to start with a list of free integers (for some set of forced integers) and turn each one of these free integers into either a gap or an element. Assuming that the number of free integers is n, the number of possibilities is 2^n, thus checking each of these possibilities for being in correspondence with the set of gaps of a numerical semigroup with PF as set of

pseudo-Frobenius numbers is unfeasible, unless n is small. Nevertheless, this naive idea can be adapted and take advantage of the already observed facts that elements can force gaps and vice-versa. Although there are examples for which fixing one integer does not force anything else, which let us expect that nothing good can result from a worst case complexity analysis, in practice it works quite well. We give some examples, but leave a detailed analysis of the complexity (perhaps average complexity) as an open problem.

The procedure we use is, for each integer v in the current list of free integers, compute all numerical semigroups containing v and the forced elements, and having the forced gaps as gaps. We proceed recursively and use backtracking when a semigroup or a dead node is found. When we revisit the node, we then suppose that v is a gap and continue with the next free integer.

Before proceeding with the pseudo-code for the recursive function RecursiveDepthFirstExploreTree that constructs the tree in a depth first manner, let us give an example which we accompany by a picture (Figure 2).
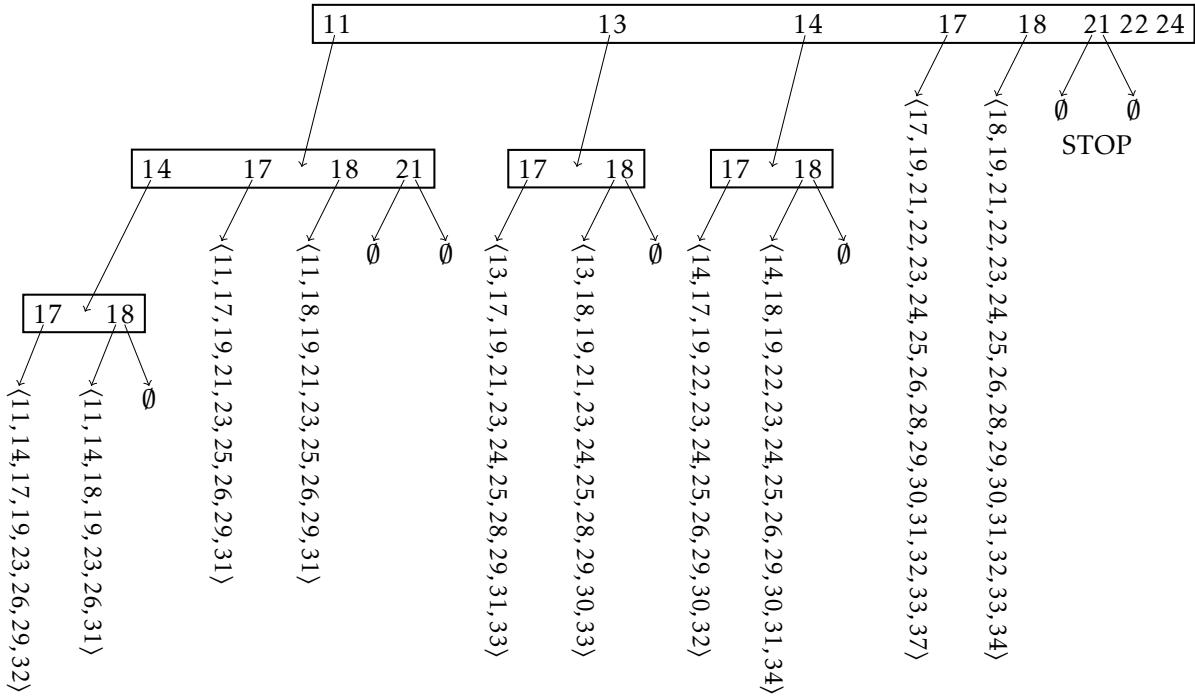
FIGURE 2. The numerical semigroups with pseudo-Frobenius numbers $\{15, 20, 27, 35\}$.

Example 35. Let PF $= \{15, 20, 27, 35\}$. From Example 24 we have a pair of lists of forced gaps and forced integers, which leaves the list

$$F = \{11, 13, 14, 17, 18, 21, 22, 24\} \tag{2}$$

of free integers.

The leaves contained in the branch that descends from 11 in Figure 2 consist of the semigroups containing 11 as forced element.

All the remaining semigroups in $\mathcal{S}(\text{PF})$ must have 11 as gap. The process then continues as Figure 2 illustrates: considering 13 as a forced integer, developing its corresponding subtree and so on.

Let us now look at the integer 21 in the root of the tree. At this point, all the semigroups in $\mathcal{S}(\text{PF})$ containing some integer preceding 21 in F have been computed. Thus, any new semigroup must have the elements in $\{11, 13, 14, 17, 18\}$ as gaps. From the computations one concludes that no new semigroup appears having 21 as an element. Thus, if some not already computed semigroup exists fulfilling the current conditions, then it must have 21 as gap. One can check that this can not happen. The remaining elements (22 and 24) need not to be checked, since 21 had to be either an element or a gap. Therefore we can stop.

Next we give pseudo-code for RecursiveDepthFirstExploreTree. The recursion ends when no more than one free integer is left. A call to the function EndingCondition is then made.

The variable **semigroups** works as a container which serves to store all the semigroups we are looking for, as they are discovered.

```
EndingCondition(g,e)                                    # g and e are such that #U \ (g∪e) ≤ 1
                           # g and e represent lists of gaps and elements, respectively
  free := ShallowCopy(Difference([1..frob],Union(g,e)));
1  if Length(free) = 0 then
2     if First(Difference(g,PF), pf → Intersection(pf + Difference(e,[0]),g) = ∅) = fail then
         Add(semigroups,NumericalSemigroupByGaps(g));
      return;
3  if Length(free) = 1 then
      if RepresentsGapsOfNumericalSemigroup(g) then
4        if First(Difference(g,PF), pf → Intersection(pf + Difference(Union(e,free),[0]),g) =
         ∅) = fail then
            Add(semigroups,NumericalSemigroupByGaps(g));

      if RepresentsGapsOfNumericalSemigroup(Union(g,free)) then
5        if First(Difference(g,PF), pf → Intersection(pf + Difference(e,[0]),Union(g,free)) =
         ∅) = fail then
            Add(semigroups,NumericalSemigroupByGaps(Union(g,free)));
      return;
```

Function EndingCondition

Lemma 36. *Function EndingCondition either does nothing or adds to **semigroups** a numerical semigroup S such that $\mathrm{PF}(S) = \mathrm{PF}$.*

Proof. It suffices to observe that any of the conditions in the "if" statements of Lines 2, 4, 5 guarantee that no forced gap outside PF can be a pseudo-Frobenius number.   □

Notice that when EndingCondition does nothing, it is because one of the following reasons:
- there is no numerical semigroup whose set of gaps is the first component of the input,
- the resulting semigroup does not have PF as set of pseudo-Frobenius numbers (it actually has more pseudo-Frobenius numbers),
- there is a free element that cannot be neither a gap nor an element.

Observe that recursion in RecursiveDepthFirstExploreTree ends (in Line 3) when there is at most one free element.

Proposition 37. *Let (fg, fe) be a pair of disjoint sets of integers contained in $U = \{0,\ldots,g_n + 1\}$. After the execution of the function RecursiveDepthFirstExploreTree with input (fg, fe), **semigroups** contains all numerical semigroups S such that $S \in \mathcal{S}(\mathrm{PF})$, $fg \subseteq \mathrm{gaps}(S)$ and $fe \subseteq \mathrm{smalls}(S)$.*

Proof. Denote by Λ the set of all numerical semigroups S such that $fg \subseteq \mathrm{gaps}(S)$, $fe \subseteq \mathrm{smalls}(S)$. We have to prove that **semigroups** $\cap \Lambda = \mathcal{S}(\mathrm{PF}) \cap \Lambda$.

$\subseteq$. It suffices to observe that the numerical semigroups are added to **semigroups** by the function EndingCondition and these belong to $\mathcal{S}(\mathrm{PF})$, by Lemma 36.

$\supseteq$. Let $S \in \mathcal{S}(\mathrm{PF})$ be such that $fg \subseteq \mathrm{gaps}(S)$ and $fe \subseteq \mathrm{smalls}(S)$. If $\#U \setminus (fg \cup fe) \in \{0,1\}$, then the function EndingCondition is called and it gives the correct output. Otherwise, we enter a recursion.

We will prove by induction on the number of free elements that the output is correct. Let us consider the smallest integer $v \in U \setminus (fg \cup fe)$. RecursiveDepthFirstExploreTree is called with input (fg, fe), and will enter the while loop, adding v to fe and computing new forced integers (SimpleForcedIntegers).

```
RecursiveDepthFirstExploreTree([fg,fe])          # fg and fe consist of forced gaps and
                                                 #   forced elements, respectively
currentfree := ShallowCopy(Difference([1..frob],Union(fg,fe)));
nfg := ShallowCopy(fg);                          # used to store new forced gaps
                                                 # without creating conflicts in the memory
  while Length(currentfree) > 1 do
      v := currentfree[1];
1     left := SimpleForcedIntegers(nfg,Union(fe,[v]));
2     if left = fail then
          right := SimpleForcedIntegers(Union(nfg,[v]),fe);
          if (right = fail) or (Intersection(right[1],right[2]) ≠ ∅) then
            break;
      else
          RecursiveDepthFirstExploreTree ([left[1],left[2]]);
      nfg := Union(nfg,[v]);
      currentfree := Difference(currentfree,[v]);
3 if Length(currentfree) ≤ 1 then
    EndingCondition (fg,fe);
```

Function RecursiveDepthFirstExploreTree

- If $v \in S$, then $left$ in Line 1 will not be equal to fail (Theorem 20), and we will call RecursiveDepthFirstExploreTree with a lager set fe, having in consequence less free integers. By induction, S is added to **semigroups**.
- Now assume that $v \notin S$. After the execution of the if-then-else starting in Line 2, v is added to the set of gaps. We have then one element less in the list of free integers, $fg \cup \{v\} \subseteq \text{gaps}(S)$ and $fe \subseteq \text{smalls}(S)$, whence the effect is the same as if we called RecursiveDepthFirstExploreTree with arguments $fg \cup \{v\} \subseteq \text{gaps}(S)$ and $fe \subseteq \text{smalls}(S)$. By induction hypothesis, S is added to **semigroups**. $\qquad\square$

Observe that if $\mathcal{S}(\text{PF}) \neq \emptyset$ and the sets fg and fe considered in Proposition 37 consist of forced gaps and forced elements, respectively, then $\Lambda \subseteq \mathcal{S}(\text{PF})$. Therefore we have proved the following corollary.

Corollary 38. *If the function RecursiveDepthFirstExploreTree is called with a pair (fg, fe), where fg consists of forced gaps and fe consists of forced elements, then at the end of its execution we have that **semigroups** = $\mathcal{S}(\text{PF})$.*

```
Input  : PF
Output: the set 𝒮(PF).
1 if (type = 1) or (type = 2 and PF[1] = PF[2]/2) then
    return IrreducibleNumericalSemigroupsWithFrobeniusNumber(frob);
2 if not (PF[type] − PF[type − 1] > PF[1]) then
    return ∅;
3 root:=ForcedIntegers (PF);
  if root = fail then
    return ∅;
  semigroups := ∅;
  RecursiveDepthFirstExploreTree(root);
  return semigroups;
```

Algorithm 3: NumericalSemigroupsWithPseudoFrobeniusNumbers

6.2. **The main algorithm.** We observe that Algorithm 3 is not efficient when there are many free integers (for some set of forced integers). A possible attempt to improve it could be to replace the recursive function by a much more efficient depth first search algorithm to parse the tree in question. Another one is to use other sophisticated theoretical tools. What is done in Appendix A could be seen as an attempt. Having at our disposal more than one approach, we can take advantage of choosing the most efficient for each situation.

Theorem 39. *Let* $\mathrm{PF} = \{g_1 < \cdots < g_n\}$ *be a set of positive integers. The output of Algorithm 3 with input* PF *is* $\mathcal{S}(\mathrm{PF})$.

Proof. Line 1 is justified by Remark 34. The justification for Line 2 comes from Corollary 10. If the stated necessary condition is not fulfilled, then $\mathcal{S}(\mathrm{PF}) = \emptyset$, which is precisely the set returned by the algorithm. As Algorithm 2 returns *fail* precisely when it is found some forced gap that is at the same time a forced element, Proposition 3 assures us that $\mathcal{S}(\mathrm{PF}) = \emptyset$, which is the set returned by the algorithm.

When nothing of the above holds, the variable **semigroups** is initialized as the empty set and it is made a call to the recursive function RecursiveDepthFirstExploreTree. As we are considering this variable global to the functions EndingCondition, the result follows from Corollary 38. $\qquad\square$

7. Running times and examples

The number of semigroups can be quite large compared to the number of free elements. The following example illustrate this possibility.

Example 40.
```
gap>  pf := [ 68, 71, 163, 196 ];;
gap> forced := ForcedIntegersForPseudoFrobenius(pf);;
gap> free := Difference([1..Maximum(pf)],Union(forced));;
gap> Length(free);
38
gap> list := NumericalSemigroupsWithPseudoFrobeniusNumbers(pf);;
gap> Length(list);
1608
```
In the continuation of previous GAP session we do a kind of verification of the result.
```
gap> ns := RandomList(list);
<Numerical semigroup>
gap> MinimalGeneratingSystem(ns);
[ 35, 38, 65, 81, 89, 94, 99, 101, 104, 106, 109, 110, 112, 113, 117, 118,
  121, 122, 133 ]
gap> PseudoFrobeniusOfNumericalSemigroup(ns);
[ 68, 71, 163, 196 ]
```

Table 3 is meant to illustrate some timings. Its content was produced by using repeatedly the commands of the first part of the previous example. The candidates to PF were obtained randomly. We observe that, although depending on some factors (such as the type or the Frobenius number we are interested in), the vast majority of the candidates would lead to the empty set. We do not consider them in this table (some were given in Table 2).

8. Random

Sometimes one may just be interested in obtaining one numerical semigroup with PF as set of pseudo-Frobenius numbers. Algorithm 3 may be too slow (it gives much information that will not be used). One could adapt the algorithm to stop once it encounters the first semigroup, but the information had to be transmitted recursively and one would end up with a slow algorithm. Next we propose an alternative (Algorithm 4), the first part of which is similar to the initial part of Algorithm 3. The main difference is in the usage of the function AnIrreducibleNumericalSemigroupsWithFrobeniusNumber instead of the function IrreducibleNumericalSemigroupsWithFrobeniusNumber, both available in the numericalsgps package. For the second part, instead of calling the recursive function, it tries to gess a path that leads

pseudo Frobenius numbers	# free	# semigroups	time
[15, 27, 31, 43, 47]	0	1	3
[16, 30, 33, 37]	9	3	12
[40, 65, 80, 89, 107, 110, 130]	5	3	29
[32, 35, 44, 45, 48]	13	7	24
[40, 65, 89, 91, 100, 106]	24	9	99
[36, 50, 56, 57, 63]	25	39	123
[43, 50, 52, 65]	35	213	605
[68, 71, 163, 196]	38	1608	16603
[38, 57, 67, 74, 79]	40	155	527
[68, 72, 76, 77]	46	177	607
[62, 78, 99, 129, 130]	53	4077	28622
[128, 131, 146, 151, 180, 216, 224, 267, 271, 287]	54	954	24253
[84, 103, 144, 202, 230, 242, 245]	56	14292	277094
[66, 85, 86, 92]	55	950	4683
[76, 79, 88, 102]	64	1409	6505
[61, 67, 94, 105]	69	4432	21471
[114, 150, 179, 182, 231, 236, 254, 321]	69	302929	7121020
[62, 73, 166, 190, 203]	77	9934	134554
[102, 104, 118, 123, 134, 146, 149]	87	15910	149910

TABLE 3. Some examples of execution data of the main algorithm.

to a leaf. Starts choosing at random a free integer v and tests its non admissibility (by checking whether SimpleForcedIntegers returns fail when called with v as if it was forced). If one does not conclude that v is non admissible, it is assumed to be a forced integer. There is an option that is part of the implementation to give a bound for the maximum number of attempts the function does. Its usage is ilustrated in Examples 41 and 42.

It may happen that no semigroup has the given set as set of pseudo-Frobenius elements, and thus the output will simply be fail.

Example 41. We look for a random numerical semigroup with PF = $\{100, 453, 537, 543\}$. The first execution of the function yields:

```
gap> pf := [ 100, 453, 537, 543 ];;
gap> ns := RandomNumericalSemigroupWithPseudoFrobeniusNumbers(pf);;time;
 MinimalGeneratingSystem(ns);
2440
[ 66, 94, 106, 126, 166, 184, 194, 206, 209, 216, 224, 230, 235, 246, 256, 263,
 267, 284, 295, 309, 363, 374, 379, 385, 391, 413 ]
```

While if we execute it a second time we get another semigroup with the desired set of pseudo-Frobenius numbers (clearly, the reader might obtain different outputs):

```
gap> pf := [ 100, 453, 537, 543 ];;
gap> ns := RandomNumericalSemigroupWithPseudoFrobeniusNumbers(pf);;time;
MinimalGeneratingSystem(ns);
7302
[ 94, 106, 123, 134, 162, 178, 184, 194, 204, 206, 222, 223, 234, 235, 238, 248,
 251, 262, 263, 266, 270, 276, 279, 283, 293, 304,  313, 336, 348, 367, 383, 415 ]
```

Example 42. If one of the free integers can neither be a gap nor an element, no semigroup exists.

```
gap> pf := [ 30, 104, 118, 147, 197, 292, 298, 315, 333, 384, 408 ];;
gap> ns := RandomNumericalSemigroupWithPseudoFrobeniusNumbers(
>  rec(pseudo_frobenius := pf, max_attempts := 100));time;
```

Input : PF, max_attempts
Output: One numerical semigroup S (at random) such that PF(S) = PF if it discovers some, fail if it
        discovers that no semigroup exists... Otherwise, suggests to use more attempts
if (type = 1) or (type = 2 and PF[1] = PF[2]/2) **then**
    └ **return** AnIrreducibleNumericalSemigroupsWithFrobeniusNumber(frob);
if not PF[type] − PF[type − 1] > PF[1]) **then**
    └ **return** fail;
f_ints := ForcedIntegers(PF);
if f_ints = fail **then**
    └ **return** fail;
$free$:= Difference([1..frob], Union(f_ints));
for $i \in [1..max\_attempts]$ **do**
    while $free \neq \emptyset$ **do**
        v := RandomList($free$);
        $nfig$:= SimpleForcedIntegers(Union(f_ints[1], [v]), f_ints[2]);
        $nfie$:= SimpleForcedIntegers(f_ints[1], Union(f_ints[2], [v]));
        if $nfig \neq$ fail **then**
            if IsRange(Union($nfig$)) **then**
            │ **return** NumericalSemigroupByGaps($nfig$[1]);
            else
            └ $free$:= Difference([1..frob], Union($nfig$));
        else
            if $nfie \neq$ fail **then**
                if IsRange(Union($nfie$)) **then**
                │ **return** NumericalSemigroupByGaps($nfie$[1]);
                else
                └ $free$:= Difference([1..frob], Union($nfie$));
            else
            └ break;

 # Info: Increase the number of attempts...

Algorithm 4: RandomNumericalSemigroupWithPseudoFrobeniusNumbers

```
fail
22
```

APPENDIX A. AN APPROACH BASED ON IRREDUCIBLE NUMERICAL SEMIGROUPS

We present here an alternative way to compute the set of numerical semigroups with a given set of pseudo-Frobenius numbers. In general, this procedure is slower than the presented above, though we have not been able to characterize when this happens. We include it the manuscript since it was the initial implementation and was used to test the other one.

A numerical semigroup is *irreducible* if it cannot be expressed as the intersection of two numerical semigroups properly containing it. It turns out that a numerical semigroup S is irreducible if and only if either PF(S) = {F(S)} or PF(S) = {F(S)/2, F(S)} (see [8, Chapter 3]). Irreducible numerical semigroups can be also characterized as those maximal (with respect to set inclusion) numerical semigroups in the set of all numerical semigroups with given Frobenius number.

The maximality of irreducible numerical semigroups in the set of all numerical semigroups with given Frobenius number implies that every numerical semigroup is contained in an irreducible numerical with its same Frobenius number. Actually, we can say more.

Lemma 43. *Let S be a numerical semigroup. There exists an irreducible numerical semigroup T such that*

(1) $F(S) = F(T)$,
(2) $]F(S)/2, F(S)[\cap PF(S) \subset T$.

Proof. Let $f = F(S)$ and let $F =]f/2, f[\cap PF(S)$. We claim that $S' = S \cup F$ is a numerical semigroup with $F(S') = f$. Take $s, s' \in S' \setminus \{0\}$.

- If both s and s' are in S, then $s + s' \in S \subseteq S'$.
- If $s \in S$ and $s' \in F$, then $s + s' \in S$, because $s' \in PF(S)$.
- If $s, s' \in F$, then $s + s' > f$, and so $s + s' \in S \subseteq S'$.

Let us show that $F(S') = f$. Assume to the contrary that this is not the case, and consequently $f \in S'$. Then as all the elements in F are greater than $f/2$, and $f \notin S$, there must be an element $s \in S$ and $g \in PF(S)$ such that $g + s = f$. But this is impossible, since all elements in $PF(S)$ are incomparable with respect to $\leq_S$ (Lemma 4).

If S' is not irreducible, then as irreducible numerical semigroups are maximal in the set of numerical semigroups with fixed Frobenius number, there exists T with $F(S) = F(S') = F(T)$ and containing S'; whence fulfilling the desired conditions. $\square$

With all these ingredients we get the following result.

Proposition 44. *Let S be a nonirreducible numerical semigroup with $PF(S) = \{g_1 < \cdots < g_k\}$, $k \geq 2$. Then there exists a chain*

$$S = S_0 \subset S_1 = S \cup \{x_1\} \subset \cdots \subset S_l = S \cup \{x_1, \ldots, x_l\},$$

with

(1) S_l irreducible,
(2) $x_i = \max(S_l \setminus S_{i-1})$ for all $i \in \{1, \ldots, l\}$,
(3) $(g_k/2, g_k) \cap PF(S) \subset S_l$,
(4) for every $i \in \{1, \ldots, l\}$, x_i is a minimal generator of S_i such that $g_j - x_i \in S_i$ for some $j \in \{1, \ldots, k\}$,
(5) for every $i \in \{1, \ldots, l\}$ and $f \in PF(S_i)$ with $f \neq g_k$, there exists $j \in \{1, \ldots, k-1\}$ such that $g_j - f \in S_i$.

Proof. Let T be as in Lemma 43. Construct a chain joining S and T by setting $S_0 = S$ and $S_i = S_{i-1} \cup \{x_i\}$, with $x_i = \max(T \setminus S_{i-1})$. Then S_i is a numerical semigroup, and x_i is a minimal generator of S_i and a pseudo-Frobenius number for S_{i-1} ([8, Lemma 4.35]). Since the complement $T \setminus S$ is finite, for some k, $S_k = T$.

Clearly, $x_1 = g_k$, and thus $x_1 - g_k = 0 \in S$. Now let $i \in \{2, \ldots, k\}$. Then $x_i \in T \setminus S$, and by Lemma 4, there exists $j \in \{1, \ldots, k\}$ such that $g_j - x_i \in S \subseteq S_i$.

Take $f \in PF(S_i) \setminus \{g_k\}$. Then $f \notin S$ and thus $g_j - f \in S$ for some $j \in \{1, \ldots, k\}$ (Lemma 4 once more). Consequently $g_j - f \in S_i$. Notice that $j \neq k$, since $f - g_k < 0$. $\square$

Given a candidate set PF of pseudo-Frobenius numbers with maximum element f, we can use the above procedure to construct from the set of all irreducible numerical semigroups with Frobenius number f, the set of all numerical semigroups having PF as a set its pseudo-Frobenius numbers. In order to compute the set of all irreducible numerical semigroups with Frobenius number f we use implementation of the procedure presented in [1] that is already part of [4]. We have slightly modified the algorithm in [1] to compute the set of irreducible numerical semigroups containing a given set of integers, and these integers are the first component of ForcedIntegers (PF). For every irreducible element in the list we then remove those minimal generators fulfilling condition (4) in Proposition 44. We add to our list of semigroups the semigroups obtained in the preceding step for which condition (5) holds, and then we proceed recursively.

Example 45. Let us illustrate the above procedure with PF = $\{10, 13\}$. The number of irreducible numerical semigroup with Frobenius number 13 is 8. However, if we first call ForcedIntegers we get:

```
gap> ForcedIntegersForPseudoFrobenius([10,13]);
[ [ 1, 2, 3, 5, 6, 10, 13 ], [ 0, 7, 8, 11, 12, 14 ] ]
```

Since we have modified the function `IrreducibleNumericalSemigroupsWithFrobeniusNumber` to output only those irreducible numerical semigroups containing $\{0, 7, 8, 11, 12, 14\}$, we obtain only two irreducible numerical semigroups: $S_1 = \langle 4, 7, 10 \rangle$ and $S_2 = \langle 7, 8, 9, 10, 11, 12 \rangle$.

For S_1 the only minimal generator that fulfills the conditions in Proposition 44 is 10. If we remove 10 from S_1, we obtain $T_1 = \langle 4, 7, 17 \rangle$, which already has the desired set of pseudo-Frobenius number.

As for S_2, again 10 is the only minimal generator fulfilling the conditions in Proposition 44, and we obtain $T_2 = \langle 7, 8, 9, 11, 12 \rangle$. This semigroup has pseudo-Frobenius number set equal to PF, and so, as with T_1 we do not need to look for new minimal generators to remove.

Thus, the only numerical semigroups with pseudo-Frobenius number set $\{10, 13\}$ are T_1 and T_2.

References

[1] V. Blanco, J. C. Rosales, The tree of irreducible numerical semigroups with fixed Frobenius number, Forum Math. **25** (2013), 1249–1261.

[2] H. Bresinsky, Symmetric semigroups of integers generated by 4 elements, Manuscripta Math. **17** (1975), 205-219.

[3] M. Delgado, "intpic", *a GAP package for drawing integers*. Available via http://www.gap-system.org/.

[4] M. Delgado, P. A. García-Sánchez and J. Morais, "NumericalSgps", *a GAP package for numerical semigroups*, *Version 0.990*; 2015. Available via http://www.gap-system.org/.

[5] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.7.7*; 2015. Available via http://www.gap-system.org/.

[6] J. Komeda, On the existence of Weierstrass points with a certain semigroup generated by 4 elements, Tsukuba J. Math. **6** (1982), 237-270.

[7] A. M. Robles-Pérez and J. C. Rosales, The genus, the Frobenius number, and the pseudo-Frobenius numbers of numerical semigroups with type two, preprint.

[8] J. C. Rosales and P. A. García-Sánchez, "Numerical Semigroups", *Developments in Maths.* **20**, Springer (2010).

[9] J. C. Rosales, P. A. García-Sánchez, J. I. García-García, and J. A. Jiménez-Madrid, Fundamental gaps in numerical semigroups, J. Pure Appl. Algebra **189** (2004), 301–313.

CMUP, Departamento de Matemática, Faculdade de Ciências, Universidade do Porto, Rua do Campo Alegre 687, 4169-007 Porto, Portugal
E-mail address: mdelgado@fc.up.pt

Departamento de Álgebra, Universidad de Granada, 18071 Granada, España
E-mail address: pedro@ugr.es

Departamento de Matemática Aplicada, Universidad de Granada, 18071 Granada, España
E-mail address: arobles@ugr.es